A DBMS FOR MOBILE TRANSACTIONS USING BI-STATE-TERMINATION

Sebastian Obermeier[1] and Stefan Böttcher[2]

[1] ABB Corporate Research, Industrial Software Systems, Segelhofstr. 1K, Baden, Switzerland
sebastian.obermeier@ch.abb.com

[2] University of Paderborn, Fürstenallee 11, 33102 Paderborn, Germany
stb@uni-paderborn.de

ABSTRACT

Whenever distributed transaction processing in MANETs or other unreliable networks has to guarantee atomicity and isolation, a major challenge is how long-term blocking of resources can be avoided in case the mobile device looses connection to other participants of the transaction. We present a new technique for treating blocked data of transaction participants that wait for a coordinator's commit decision. Our technique, Bi-State-Termination (BST), gives participants that have moved during transaction execution the possibility to continue transaction processing before they know the coordinator's decision on transaction commit. The key idea of our technique is to consider both possible outcomes (commit and abort) of unknown transaction decisions. Within this paper, we describe a fast implementation of the fundamental relational database operations for a DBMS supporting the BST transaction synchronization protocol that avoids long-term transaction blocking.

KEYWORDS

Mobile Database Transaction Processing, Transaction Blocking, Bi-State-Termination

1. INTRODUCTION

With growing interest in mobile ad-hoc networks and increasing capabilities regarding processing power and connectivity, an interesting and important challenge is to combine database technology with mobile devices. However, a transfer of traditional research results to mobile devices is hindered by problems like message loss, unpredictable disconnections of mobile devices, and network partitioning. When applying database technology that was designed for traditional fixed-wired networks in such mobile environments, the problem of long or even infinitely long blocking times for commonly accessed resources arises. For instance, a message that releases blocked resources may never reach its destination and leads to blocking of resources longer than intended.

Traditional transaction processing is unsuitable for such a scenario for the following reason. It would prevent participants that have moved during the atomic commit protocol execution and therefore have not received the transaction's commit decision from using parts of their own data in concurrent transactions. In mobile networks, there is no guarantee that these moved participants will ever receive the transaction's decision. Therefore, traditional transaction processing would cause resources to be unusable for an infinitely long period of time.

1.1. Problem Description

Atomic commit protocols (ACPs) like 2-Phase-Commit (2PC, [9]), 3-Phase-Commit (3PC, [22]), and consensus based protocols (Paxos Consensus, [10]) are used to guarantee an atomic execution of distributed

transactions in fixed wired networks. However, when transaction support is required for mobile ad-hoc environments, problems like network partitioning, node disconnection, and node movement may lead to the case where databases do not receive the atomic commit protocol decision and block. According to [18], we can distinguish two kinds of blocking:

Definition 1 *Atomic commit protocol blocking* occurs, if during an execution of an atomic commit protocol for a transaction T, an arbitrary sequence of failures leads to a situation where the atomic commit protocol instance cannot terminate with a unique commit or abort decision d.

Several proposals, e.g. [10, 20], tackle the problem of atomic commit protocol blocking by using multiple coordinators. Therefore, we do not focus on the problem of protocol blocking.

Definition 2 *Transaction Blocking:* A transaction T is *blocked*, after a database proposed to execute T (e.g. by sending a voteCommit message) and waits for the final commit decision, but is not allowed to abort or to commit T unilaterally on its own.

Transaction blocking summarizes the unilateral impossibility to abort or commit a transaction, but does *not* mean that a transaction U waits to obtain locks from a concurrent transaction, since in this case U can be aborted by the database itself. Even time-out based approaches (e.g. "my commit vote is valid until 3:23:34") cannot solve the problem of transaction blocking, since this would fulfill the requirements of the coordinated attack scenario [9], in which a commit decision is not possible under the assumption of message loss.
The problem of *infinite transaction blocking* for a transaction T occurs, if a database has sent a voteCommit message on T, but will never receive the final commit decision, e.g. due to disconnection, movement, or network partitioning. Whenever the database can still communicate, e.g. with the user, the blocked data of T will prevent concurrent and conflicting transactions U from being processed.

1.2. Contributions

The main contributions of this paper are:

- It describes BST, a transaction termination mechanism for mobile transactions in unreliable environments that solves the infinite transaction blocking problem, i.e. each database maintains control of its resources during the whole transaction execution.
- It outlines how and proves that BST in combination with 2-Phase-Locking [8] guarantees serializability and atomicity.
- We give experimental results that demonstrate the feasibility of Bi-State-Termination in environments where the commit decision may get lost.

Beyond our previous contributions [18, 19], this paper further

- outlines how recovery from database main memory failures can be done with BST, and how BST can be used as a log book,
- describes three BST implementations, one being a complete in-memory solution, one being a disk-base solution, and one focusing on processing speed,
- outlines implementation details of the fastest variant regarding read and write operations,
- describes an evaluation comparing the presented BST implementations regarding transaction processing time in case of blocked transactions.

2. TRANSACTION MODEL

BST is a protocol that can be applied to databases that participate in an atomic commit protocol independent of whether or not the other databases that also participate in the same global transaction use the same BST protocol.
During the execution of a transaction, a database enters the following phases: the *read-phase*, the *commit decision phase*, and, in case of successful commit, the *write-phase*. While executing the read-phase, each transaction invokes necessary sub-transactions and carries out write operations on its private transaction storage only. During the commit decision phase, the participating databases use an *atomic commit protocol* to decide on the transaction's commit decision. During the atomic commit protocol, participants vote for

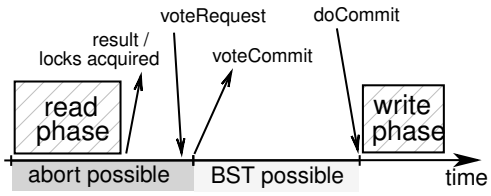

Figure 1. 2PC Transaction Execution

"commit" or "abort". A database that voted for "commit" is not allowed to abort or commit the transaction on its own until it has received the coordinator's commit decision. A participant that voted for "abort" can immediately abort the transaction, which means, the database must restore a state that is equal to a situation in which the transaction has never been executed, i.e. all changes caused by the aborted transaction must be rolled back.
If the transaction's outcome commit decision, which can be either *commit* or *abort*, is commit, the database executes the sub-transaction's write phase. During this phase, the private transaction storage is transferred to the durable database storage, such that the changes done throughout the read-phase become visible to other transactions after completion of the write-phase. If the outcome is abort, all involved databases abort their corresponding sub-transactions.

Definition 3 Two operations O_i and O_j conflict $\Longleftrightarrow \exists$ tuple $t \exists$ attribute $a : (O_i$ accesses $t.a$ $\wedge\ O_j$ accesses $t.a \wedge ((O_i$ writes $t.a) \vee (O_j$ writes $t.a)))$

Definition 4 A transaction T_j *depends* on T_i if and only if on a database D at least one operation O_i of T_i conflicts with an operation O_j of T_j, and O_i precedes O_j.

Definition 5 The *serialization graph* of a set of transactions contains the transactions as nodes and a directed edge $T_i \rightarrow T_j$ for each pair (T_i, T_j) of transactions for which T_j depends on T_i.

Definition 6 *Serializability* requires the serialization graph of all committed transactions to be acyclic.

Definition 7 Assume a database in state S_0 executes a transaction T_i. Then, we call $\mathsf{Result}_{T_i}(S_0)$ the result value that the databases returns after finishing the read phase of T_i.

Above the time line, Figure 1 shows the standard application of 2PC for a distributed transaction T_i when a locking mechanism like 2-Phase-Locking [8] is used for transaction synchronization. Below the time line, Figure 1 illustrates the possible database reactions in case the database does not receive expected messages after a timeout. As long as the database has not voted for commit, it can still abort T_i and release the locks. Different from traditional 2PC, *Bi-State-Termination* (BST) allows a database to terminate a transaction even after the commit vote has been sent and before a doCommit or doAbort command has been received. The transaction execution shown in Figure 1 involves two messages showing that the so-called *lock point* was reached: Both the voteRequest message and the doCommit message indicate that all databases have obtained all necessary locks. However, for the purpose of ensuring serializability, it is only necessary to reach this lock point once, as atomic commit protocol optimizations like "unsolicited vote optimization" [23] show.
As the transaction sequence is fixed after the voteRequest message has been received by each database, serializability is guaranteed. In other words, serializability does not require to hold any lock longer than until the database receives the voteRequest command. However, for ensuring strictness, i.e. to guarantee recoverability and to avoid cascading aborts, each database must hold all locks until the doCommit message is received and the write phase has been finished. Unfortunately, if the doCommit message is lost or cannot reach a database, the database must hold the locks and cannot abort the transaction on its own, which however, is possible before the vote for commit message has been sent.
In the remainder of this section, we develop a solution that not only unblocks and processes concurrent and depending transactions if the commit decision cannot be received by a database for a longer period of time. Our solution, which is called *Bi-State-Termination* (BST) of a transaction T_i, also guarantees atomicity.

The main idea of BST is that if the coordinator's decision for a transaction T_i is delayed, a concurrent transaction T_c depending on T_i can be processed by transferring the required locks from T_i to T_c, and then by executing T_c on two database states: one state having T_i committed, the other state having T_i aborted. However, it is the database's choice whether or not and after which timeout it applies BST.
We want to achieve that all transactions belonging to a distributed global transaction are executed in an atomic fashion, and that each concurrent execution of different distributed global transactions is serializable, i.e. the execution produces the same output and has the same effect on the databases as some serial execution of the same distributed global transactions.

Definition 8 Assume that a database is in a state S_0 that has been created by some previous transactions before the write phase of T_i is executed. Assuming no concurrent transaction has changed the database state while T_i is executed, we call the database state that is caused by the write phase of T_i the state S_{T_i}. Let T_i consist of the sequence of operations $\mathrm{O}_{i_1}, \mathrm{O}_{i_2}, \ldots, \mathrm{O}_{i_n}$. When this sequence of operations is applied to the database, we call the changes that have been made by the operations the *delta of the transaction* T_i. We write $\Delta_{T_i}(S_0)$ if the sequence of operations $\mathrm{O}_{i_1}, \ldots, \mathrm{O}_{i_n}$ is applied on the database state S_0. When T_i has been committed, the result of applying Δ_{T_i} to the database state S_0 becomes visible for other transactions, thus we get the new database state S_{T_i}, for which we write $S_{T_i} = S_0 \oplus \Delta_{T_i}(S_0)$.

Note, that when a transaction T_i, which, for example, increments integer values, is executed on two different database states S_0 and S_1, the transaction execution can lead to different deltas $\Delta_{T_i}(S_0)$ and $\Delta_{T_i}(S_1)$. However, when T_i does not include branches or loops, the sequence of operations remains the same for all executions.

Lemma 9 *Assume a database is in state S_0, a transaction T_i is executed, and a concurrent transaction T_j is started, but T_i does not depend on T_j and vice versa. Therefore, the changes of transaction T_j do not affect the execution of the transaction T_i. The equations*

$$\Delta_{T_j}(S_0) = \Delta_{T_j}(S_{T_i}) \quad \textit{and} \quad \Delta_{T_i}(S_0) = \Delta_{T_i}(S_{T_j})$$

hold, which means that the modifications of a transaction T_j are independent of any previous modification of a non-dependent transaction T_i and vice versa.

Proof Assume the database state before the execution of T_i and T_j is S_0.

$$\begin{array}{ll}
 & T_j \text{ does not depend on } T_i \text{ and} \\
 & T_i \text{ does not depend on } T_j \\
\iff & \forall \text{ tuple } t \, \forall \text{ attribute } a \, (\neg O_i \text{ accesses } t.a \\
 & \vee \neg O_j \text{ accesses } t.a \vee (O_i \text{ reads } t.a \wedge O_j \text{ reads } t.a)) \\
 & \qquad (\text{follows from Def. 3 and 4}) \\
\implies & S_0 \oplus \Delta_{T_i}(S_0) \oplus \Delta_{T_j}(S_{T_i}) \\
 & = O_{j_n}(\ldots O_{j_2}(O_{j_1}(O_{i_n}(\ldots O_{i_1}(S_0))))) \\
 & = O_{i_n}(\ldots O_{i_2}(O_{i_1}(O_{j_n}(\ldots O_{j_1}(S_0))))) \\
 & \qquad (\text{since operations do not conflict}) \\
 & = S_0 \oplus \Delta_{T_j}(S_0) \oplus \Delta_{T_i}(S_{T_j}) \\
\implies & \left(\Delta_{T_j}(S_0) = \Delta_{T_j}(S_{T_i})\right) \wedge \left(\Delta_{T_i}(S_0) = \Delta_{T}(S_{T_j})\right)
\end{array}$$

Therefore, whenever a set BT of transactions is blocked, the result of a transaction T may only be influenced by those transactions $DBT \subseteq BT$ on which T is dependent.

3. SOLUTION

In the following, we focus on the question:

> What can a database D executing a transaction T_i that is blocked do, when a concurrent transaction T_c requests access to data tuples accessed by T_i in a conflicting way.

One proposed solution to answer this question can be found in standard literature for databases, and is quite simple: Let T_c wait until T_i has released its locks and is completed. However, during the waiting, the

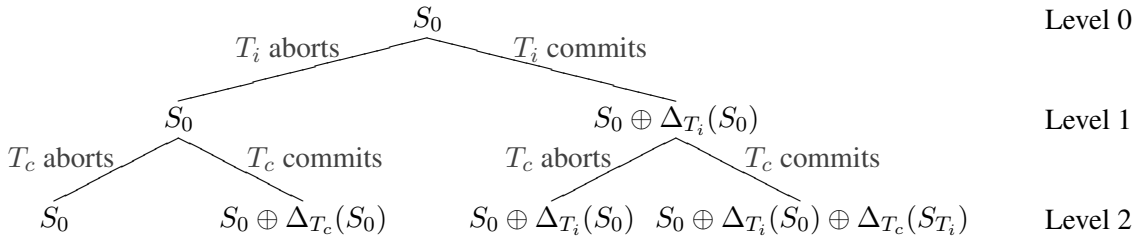

Figure 2. Possible database states if T_c depends on T_i, and T_i blocks

number of transactions that wait concurrently may increase if the blocking continues. Furthermore, if T_c is a distributed transaction too, this may cause further transactions to wait on other databases, too. Another possibility is to abort the concurrent transaction T_c. Although correct, this behavior is not satisfying.

Our solution called *Bi-State-Termination* is based on the following observation: Whenever transaction blocking occurs, the database does not know whether a transaction T_i waiting for the commit decision will be aborted or committed. However, only if the transaction is committed, the database state changes. Let S_0 denote the database state before T_i was executed. Although the database does not know the commit decision for T_i, it knows for sure that either S_0 or $S_0 \oplus \Delta_{T_i}(S_0)$ is the correct database state, depending on the commitment of T_i. Figure 2 shows these two possible states in the tree. With this knowledge, the database can try to execute a concurrent transaction T_c that depends on T_i on both states S_0 and $S_0 \oplus \Delta_{T_i}(S_0)$. Whenever the two executions of T_c on S_0 and on $S_0 \oplus \Delta_{T_i}(S_0)$ return the same results to the Initiator, i.e. $\mathsf{Result}_{T_c}(S_0) = \mathsf{Result}_{T_c}(S_{T_i})$, T_c can be committed regardless of T_i, even though T_c depends on T_i. Otherwise, it is the application's choice whether it handles two possible transaction results. However, since T_c depends on T_i, we might have $\Delta_{T_c}(S_0) \neq \Delta_{T_c}(S_{T_i})$. Therefore, the database must store both deltas $\Delta_{T_c}(S_0)$ and $\Delta_{T_c}(S_{T_i})$, and if T_c commits before T_i is committed or aborted, the database knows that either the state $S_0 \oplus \Delta_{T_c}(S_0)$ is valid, or the state $S_0 \oplus \Delta_{T_i}(S_0) \oplus \Delta_{T_c}(S_{T_i})$ is valid.

Figure 2 shows the execution tree different of a different situation, i.e. with both T_i and T_c being blocked. The leaves represent the database states that may be valid depending on the decisions for the blocked transactions T_i and T_c.

3.1. Bi-State-Termination

Let $\Sigma = \{S_0, \ldots, S_k\}$ be the set of all legal possible database states for a database D. A *traditional transaction* T_i is a function $T_i : \Sigma \mapsto \Sigma$, $S_a \rightarrow S_b$, which means the resulting state S_b of T_i depends only on the state S_a on which T_i is executed.

A *Bi-State-Terminated transaction* T_i is a function $BST : 2^{\Sigma} \mapsto 2^{\Sigma}$,

$$\underbrace{\{S_i, \ldots, S_j\}}_{\text{Initial States}} \rightarrow \underbrace{\{S_i, \ldots, S_j\}}_{T_i \text{ aborts}} \cup \underbrace{\{T_i(S_i), \ldots, T_i(S_j)\}}_{T_i \text{ commits}}$$

that maps a set $\Sigma_{\text{Initial}} \subseteq \Sigma$ of Initial States to a super set $\Sigma_{\text{Initial}} \cup \{T_i(S_x) | S_x \in \Sigma_{\text{Initial}}\}$ of new states, where $T_i(S_x)$ is the state that is reached when T_i is applied to S_x.

This concept of Bi-State-Termination leads to the following commit decision rules for the transaction execution of a transaction T_i on a database *DB*:

DB checks T_i's dependencies on concurrent blocking transactions. The following situations may occur: T_i is independent of all currently blocked transactions. Then, T_i can be executed immediately.

Otherwise, T_i depends on a set $\{T_1 \ldots T_n\}$ of blocked transactions. As T_i depends on each of the transactions $\{T_1 \ldots T_n\}$, each of them has reached its lock point before T_i. Thus, for any concurrent execution of $\{T_1 \ldots T_n\}$, there is an equivalent serial execution *ESE* of $\{T_1 \ldots T_n\}$. As *ESE* only has to reflect the order in which transactions leave the lock point, *ESE* can always be constructed, as described below. Thus, serializability is guaranteed for $\{T_1 \ldots T_n\}$. Then, *DB* can *Bi-State-Terminate* the transactions $\{T_1 \ldots T_n\}$, i.e. *DB* can execute the transaction T_i on all possible combinations of abort and commit decisions of the transactions $\{T_1 \ldots T_n\}$ in *ESE*.

The serializable sequence *ESE* of the transactions $\{T_1 \ldots T_n\}$, on which T_i is executed, must obey the following conditions. For each pair (T_j, T_k) of the transactions for which a dependency $T_j \rightarrow T_k$ exists, T_j left its lock point before T_k left its lock point. However, in order to execute the transaction T_k that depends

on the blocked transaction T_j, the transaction T_j must have been Bi-State-Terminated. In this case, T_j has completed all of its operations before T_k has started. Thus, T_j is executed before T_k within *ESE*.
Note that if there is no dependency $T_j \rightarrow T_k$, and no dependency $T_k \rightarrow T_j$, the execution sequence of the blocked transactions (T_j, T_k) does not matter since the transactions are independent of each other, cf. Lemma 9.
This means, the transaction T_i must at most be executed on all combinations of commit and abort decisions for the blocked transactions $T_1 \ldots T_n$, but only on one sequence (permutation) of the transactions $T_1 \ldots T_n$. If T_i is executed on multiple database states, this might yield to different results. Let $S_1 \ldots S_{2^n}$ be the states that can be reached by any combination of commit/abort decisions on the n transactions $\{T_1 \ldots T_n\}$ that are Bi-State-Terminated. If $\mathsf{Result}_{T_i}(S_0) = \ldots = \mathsf{Result}_{T_i}(S_{2^n})$ holds, transaction T_i can be committed since it has a unique result. Otherwise, the application that initiated T_i can choose whether

- it aborts T_i completely,
- it commits T_i and deals with multiple possible results,
- it aborts or commits T_i only for some commit/abort combinations of Bi-State-Terminated transactions, called execution *branches*. For example, the application may specify that T_i should only commit when T_k aborts, and that T_i should abort otherwise,
- it waits.

When T_i or a single execution branch of T_i commits, the database merges the corresponding delta of T_i with the possible branch state.

Example 10 *Assume T_c depends on T_i, but, different from Figure 2, T_c shall only commit when T_i commits and otherwise abort. As illustrated in Figure 3, a commit of T_c would only involve the left-most and rightmost branches of Figure 2. Therefore, $\Delta_{T_i}(S_0)$ in Level 1 of Figure 2 is replaced with $(\Delta_{T_i}(S_0) \oplus \Delta_{T_c}(S_{T_i}))$ in Level 1 of Figure 3 since, in this case, a commit of T_i automatically means a commit of T_c. Note that in this example, the commit* decision *for T_c is made* before *the decision of T_i, but the* execution *sequence is the other way round, namely $T_i < T_c$. Furthermore, the tree of Figure 3 is flattened one level compared to Figure 2 since only T_i is yet blocked.*

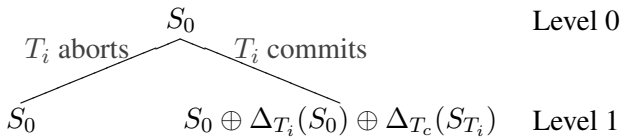

Figure 3. T_c should commit only if T_i commits

3.2. Complexity

lt can be seen that the complexity of the Bi-State-Termination of T_i depends on the number of blocked transactions b, and that BST has a complexity of $O(2^b)$ database states. However, our implemented solution uses a compact data structure and optimizes read and write operations in such a way that each transaction operation must only be executed once, regardless of the number of blocked transactions. Although, in the worst case, the number of tuples may grow exponentially, standard database query optimization techniques can be fully applied. Furthermore, a transaction still can wait instead of being executed on too many states.

3.3. Correctness

Theorem 11 *Bi-State-Termination in combination with 2-Phase-Locking guarantees serializability.*

Proof As our solution uses Two-Phase Locking (2PL) and 2PL is proven to guarantee serializability according to [5], we show that Bi-State-Termination does not change the order of pairs of conflicting operations of transactions given by 2PL and therefore guarantees serializability, too: Our transaction execution involves one point, namely the lock point, where each transaction that belongs to a global transaction must hold all locks. This means that the request to vote on a transaction T's commit status can only be sent by the coordinator after all databases acquired the necessary transaction locks for their sub-transactions $T_1 \ldots T_n$

of T, which each database indicates by sending the sub-transaction's result. The sequence of transactions is fixed at that time when each transaction enters its lock point and is not changed by BST. Although Bi-State-Termination may release locks after this lock point, the release of locks does not change the order of comitted transactions for the following reason. A transaction T_c that gets locks from a Bi-State-Terminated transaction T_i is either executed after T_i has been committed ($T_i < T_c$) or T_i is aborted.

Note that although the commit command for T_c may be issued before the commit command of T_i, the order of applying the transactions on the database is still $T_i < T_c$.

4. BST REWRITE RULES

Our BST rewrite rule system modifies each database relation in such a way that it gets an extra column *"Conditions"* that describes for each database tuple the condition under which it is regarded as being true. Furthermore, the database contains a single table *"Rules"* storing rules that relate these conditions to each other.
Whenever currently active transactions insert, delete or update tuples in a relation R, whether or not a tuple t will finally belong to R depends on the commit or abort decision of these transactions. We use a condition in order to express which of the active transactions must commit and which must abort, such that a tuple t finally belongs to a relation R. In our implementation, each relation R is augmented by an extra column "Conditions" that, for each tuple t, stores the condition under which it finally belongs to R.
This can be implemented by the following rewrite rule that modifies the create table command for database relations:

```
create table R ( <column definitions> )
   ⇒ create table R'( <column definitions, string conditions>)
```

4.1. Status Without Active Transactions

When all transactions are completed either by commit or by abort, the column "Conditions" contains the truth value "true" for each tuple in each relation of the database. The truth value "true" represents the fact that the tuple belongs to the relation without any further condition about the commit status of an active transaction.

4.2. Write Operations on the BST Model

4.2.1. Insertion

Whenever a tuple $t = (\text{value}_1, \ldots, \text{value}_N)$ is inserted into a relation R by a transaction with transaction identifier T_i, we implement this by inserting $t' = (\text{value}_1, \ldots, \text{value}_N, T_i)$ into the relation R', i.e. the database system implementation applies a rewrite rule:

insert into R values (value$_1$,. . .,value$_N$)
⇒ insert into R' values (value$_1$,. . .,value$_N$, T_i).

The idea behind the value T_i stored in the condition column of R' is to show that the tuple t belongs to the database relation R if and only if transaction T_i will be committed.

4.2.2. Deletion

Whenever a tuple $t = (\text{value}_1, \ldots, \text{value}_N)$ is deleted from a relation R by a transaction with transaction identifier T_i, we look up the tuple $t' = (\text{value}_1, \ldots, \text{value}_N, C) \in R'$ representing the tuple $t \in R$, where C is the condition under which t belongs to the database relation R.
We implement the deletion of t from R by the transaction T_i by replacing the condition C found in t' with a condition C_2 and by adding a logical rule to the table *Rules* stating that C_2 is true if and only if C is true and T_i is aborted. For this purpose, the database system applies the following rewrite rule, where $A_1, \ldots, A_N$ denote the values $(\text{value}_1, \ldots, \text{value}_N)$ for the attributes of R:

delete t from R where t.A$_1$=value$_1$, . . .,t.A$_N$=value$_N$
⇒ update t' in R' where t.A$_1$=value$_1$, . . .,t.A$_N$=value$_N$ set condition=C$_2$;
insert into rules values (C$_2$, C$_1$ and not T_i)

The idea behind this rewriting is the following. (not T_i) represents the condition that transaction T_i will be aborted. The inserted rule states that C_2 is true if C_1 is true and T_i will be aborted. After the update operation, we have a tuple $t' = (\text{value}_1, \ldots, \text{value}_N, C_2)$ in R' which represents the fact that t belongs to R if and only if C_2 is true, i.e. if C is true and T_i is aborted.

4.2.3. Update

An update of a single tuple is simply executed as a delete operation followed by an insert operation.

4.2.4. Set-Oriented Write Operations

When a transaction inserts, updates, or deletes multiple tuples within a single operation, this can be implemented by a collection of individual insert, update, or delete operations.

4.2.5. Completion of a Transaction

When transaction T_i is completed with commit, the condition T_i is replaced with true in each rule in the Rules table and in each value found in the column "Conditions" of a relation R'. However, when T_i is completed with abort, T_i is replaced with *false* in each rule found in the Rules table, and each tuple of R' containing the value T_i in the column "Conditions" is deleted.
Furthermore, rules that contain the truth value *true* or *false* are simplified. Whenever this results in a rule (C, *true*) or in a rule (C, *false*), then C itself is replaced with the value "true" or "false" respectively. Other rules that contain C are simplified as well. Furthermore, all tuples t' in which C occurs are treated as follows. If the rule is (C, *true*), the value C is replaced with true in each tuple t' in which C occurs in the column "Conditions". However, if the rule is (C, *false*), each tuple t' in which C occurs in the column "Conditions" is deleted. Finally, rules (C, *true*) or (C, *false*) are deleted from the relation *Rules*.

Example

Consider a database with a relation R that only consists of the attribute "Name" and initially stores only a single data record with the value "Mitch". On R, the database executes the sequence $T_1 < T_2 < T_3$ of transactions:

T_1: insert "Miller"
T_2: delete "Mitch"
T_3: change "M" to "R" in each name

Line	Name	Condition	Comment
1	Mitch		Initial
2	Mitch		Content after BST of T_1
3	Miller	C_1	
4	Mitch	C_2	Content after BST of T_1, T_2
5	Miller	C_1	
6	Mitch	C_3	Content after BST of T_1, T_2, T_3
7	Miller	C_4	
8	Ritch	C_5	
9	Riller	C_6	

Table 1. Content after Bi-State-Terminating T_1, T_2, and T_3

Line 1 of Table 1 represents the initial database state S_0 of R', lines 2-3 show the content of R' after BST of T_1, lines 4-5 represent the table content after BST of T_1 and T_2, while lines 6-9 show the table after BST of T_1, T_2, and T_3. The conditions C_i in the column "Condition" of Table 1 are linked to the "Rules" column of Table 2. Table 2 defines for each condition C_i by a boolean formula composed of other conditions and/or elementary conditions T_j, $\overline{T_k}$, where T_j in the column "Definition" of Table 2 represents that transaction T_j will commit and $\overline{T_k}$ represents that T_k will abort. When C_i is valid, the row (<t>, C_i) of Table 1 represents that the tuple <t> is in R. The condition C_4, for example, is fulfilled when T_1 commits and T_3 aborts. In this case, line 7 of Table 1 becomes valid.

Condition	Definition	Comment
–	–	Initial
C_1	T_1	Content after BST of T_1
C_1	T_1	Content after BST of T_1, T_2
C_2	$\overline{T_2}$	
C_3	$\overline{T_2}\overline{T_3}$	Content after BST of T_1, T_2, T_3
C_4	$T_1\overline{T_3}$	
C_5	$\overline{T_2}T_3$	
C_6	T_1T_3	

Table 2. Rules Table after Bi-State-Terminating T_1, T_2, and T_3

4.3. Read Operations on the BST Model

Whenever a read operation on R is implemented by a read operation on R', the conditions are kept as part of the result. The relational algebra operations are implemented as follows.

4.3.1. Selection

Each selection with selection condition SC that a query applies to a relation R, will be applied to R', i.e. the database system applies the following rewrite rule to each selection:

SC(R) $\Rightarrow$ SC(R')

4.3.2. Duplicate Elimination

Duplicate elimination is an operation that is used to implement projection and union. When duplicates occur, their conditions are combined with the logical OR operator. That is, given the relation R' contains two tuples $t'_1 = (\text{value}_1, \ldots, \text{value}_N, C_1)$ and $t'_2 = (\text{value}_1, \ldots, \text{value}_N, C_2)$ these two tuples are deleted and a single tuple $t' = (\text{value}_1, \ldots, \text{value}_N, C_{C12})$ is inserted into R, and a rule (C_{C12}, C_1 or C_2) is inserted into the Rules table.

4.3.3. Set Union

Set union of two relations R_1 and R_2 is implemented by applying duplicate elimination to the set union of R'_1 and R'_2. The database system applies the following rewrite rule:

$R_1 \cup R_2 \Rightarrow$ removeDuplicates($R_1' \cup R_2'$)

4.3.4. Projection

Projection of a relation R_1 on its attributes $A_1, \ldots, A_N$ is implemented by applying duplicate elimination to the result of applying the projection to R'_1 including the column "Conditions". The database system applies the following rewrite rule:

P($A_1, \ldots, A_n$) (R_1) $\Rightarrow$ removeDuplicates(P($A_1, \ldots, A_n$, conditions) (R'_1))

4.3.5. Cartesian Product

Whenever the cartesian product $R_1 \times R_2$ of two relations R_1 and R_2 must be computed, this is implemented using R'_1 and R'_2 as follows. For each pair (t'_1, t'_2) of tuples $t'_1 = (\text{value}_1, \ldots, \text{value}_N, C_1)$ of R'_1 and $t'_1 = (\text{value2}_1, \ldots, \text{value2}_N, C_2)$ of R'_2, a tuple $t'_{12} = (\text{value}_1, \ldots, \text{value}_N, \text{value2}_1, \ldots, \text{value2}_N, C_{C12})$ is constructed and stored in $(R_1 \times R_2)'$. The database system applies the following rewrite rule:

$R_1 \times R_2 \Rightarrow (R_1 \times R_2)'$
where $(R_1 \times R_2)'$ can be derived by computing the set
$\{ (t_1, t_2, C_{C12}) \mid (t_1, C_1) \in R_1' \text{ and } (t_2, C_2) \in R_2' \}$
and by adding a rule (C_{C12}, C_1 and C_2) for each pair of C_1 and C_2 to the Rules table.

4.3.6. Set Difference

Whenever the set difference $R_1 - R_2$ of two relations R_1 and R_2 must be computed, this is implemented using R_1' and R_2' as follows. The set difference contains all tuples $t_1' = (\text{value}_1, \ldots, \text{value}_N, C_1)$ of R_1' for which no tuple $t_2' = (\text{value2}_1, \ldots, \text{value2}_N, C_2)$ of R_2' exists, and furthermore, it contains a tuple $t_{12}' = (\text{value}_1, \ldots, \text{value}_N, C_{C12})$ for each tuple $t_1' = (\text{value}_1, \ldots, \text{value}_N, C_1)$ of R_1' for which a tuple $t_2' = (\text{value2}_1, \ldots, \text{value2}_N, C_2)$, $C_2 \neq C_1$, of R_2' exists. The condition C_{C12} is *true* if and only if (C_1 and not C_2) is *true*. The database system applies the following rewrite rule:

R_1 - $R_2 \Rightarrow$ R_1' - R_2'
where (R_1' - R_2') can be derived by computing the union of the following sets S_1 and S_2:
S_1 = { (t_1,C_1) | exists (t_1,C_1) $\in$ R_1' and not exists C_2 such that (t_1,C_2)$\in$ R_2' }
S_2 = { (t_1,C_{C34}) | exists (t_1,C_3)$\in$ R_1' and exists (t_1, C_4)$\in$ R_2' such that $C_3 \neq C_4$}
and by adding a rule (C_{C34}, C_3 and not C_4) for each pair of C_3 and C_4 used in S_2 to the Rules table.

4.3.7. Other Algebra Operations

Other operations of the relational algebra like join, intersection, etc. can be constructed by combining the implementation of the basic operations. Of course, query optimization of operations like join etc. is possible.

5. DATABASE CONSTRAINTS

We classify two kinds of constraints: *integrity constraints* and *consistency constraints*. Following [3], integrity constraints are defined as data tuple characteristics that are independent of other data tuples. Thus, integrity constraints can be checked row-by-row. Consistency constraints, as defined in [8], specify relationship characteristics of two or more data tuples. Thus, consistency constraints can be formulated as read queries that return boolean values. We assume that before a constraint check is performed, the database is in a valid state, i.e. it's integrity and consistency constraints are fulfilled. Thus, a transaction converts the database from one valid state to another valid state, and we only have to check whether the changes of a transaction violate the defined constraints.

5.1. Integrity Constraints

One characteristics of BST is that the data accessed by write operations is *already* present in the database before the transaction is committed, and that the data accessed by pending delete operations is *still* present. Furthermore, each commit/abort combination of Bi-State-Terminated transactions is possible. Thus, integrity constraints can be checked for each database row without obeying the conditions relation. Whenever a data item violates an integrity constraint, the transaction must be aborted, otherwise the database can vote for commit.

5.2. Consistency Constraints

Consistency constraints formulated as boolean queries are checked as follows. The query is evaluated and eventually returns different result values including conditions representing validity, for example R_Q =((true), C_k), whereas C_k can be composed of different condition formulas. Whenever a consistency constraint may be violated, i.e. we get a result value ((false), C_k), we furthermore check whether the composed condition formula C_k is a *contradiction*. If C_k is not a contradiction, the consistency constraint may be violated and the database votes for abort. If the database gets ((true), C_k) and ((false), C_l) values and all C_l are a contradiction, the database votes for abort since the consistency constraint is fulfilled and cannot be violated.

Referential Integrity Constraints

For referential integrity, we join the primary and foreign key, eliminate duplicates, and derive the corresponding condition formulas for each joined data item. Then, we check for each joined tuple whether the formulated referential integrity constraint may be violated.

Example 12 *Assume the relation* Order *contains IDs of tuples of the relation* Customer*. A referential integrity constraint C assures that each order contains an ID of a customer listed in relation* Customer*, i.e.*

	ID	Attributes	Conditions
(1)	1	α_1	
(2)	2	α_2	

Table 3. R" is derived from the table R by adding the column "Conditions"

$\forall o \in$ *Order* $\exists c \in$ *Customer*: $o.$*CID* $= c.$*CID*. *Assume, a referential integrity constraint RC results in the constraint value R_t =((true), $C_1 \wedge \overline{C_6} \Rightarrow (C3 \wedge C2) \vee \overline{C_6}$) for a certain data item, which indicates that the referential integrity is fulfilled when the formula R_t is true. Since R_t is a tautology, the constraint is fulfilled regardless of which Bi-State-Terminated transactions commit or abort.*

Whenever a database constraint may be violated by a transaction, the transaction cannot vote for commit and thus is aborted.

6. FAILURE RECOVERY

The use of the Rules table allows implementing failure recovery in the following different way. A log book for log-based failure recovery is not necessarily required for undoing or redoing a transaction for the following reason. Whether or not a transaction is committed or aborted or currently running can be checked by investigating the Rules table. An atomic write of the commit record to a log book can be replaced by an atomic write operation on the Rules table which adds a rule ('T_i', *true*) if transaction T_i commits or ('T_i', *false*) if transaction T_i aborts, before the entries 'T_i' are replaced with *true* or *false* respectively elsewhere. After all simplification steps have been applied to the Rules table and to the augmented relations R_i' and after the distributed commit decision has been acknowledged by all transaction partners, the rule entry ('T_i', *true*) or ('T_i', *false*) is not needed anymore and can be deleted from the Rules table. This approach is as safe against power supply failures as log book-based approaches, because whether or not a transaction 'T_i' is completed can be seen by looking for an entry ('T_i', ...) in the Rules table, and all the derivation steps on the Rules table and in the database relations R_i' can be reconstructed from a given entry ('T_i', *true*) or ('T_i', *false*) in the Rules table.

7. IMPLEMENTATION

We have implemented BST in three versions and have compared their performance using a stress test. The first implementation, called *BST-Disk*, uses the Rules table and rewrite rules as stated in Section 4, and stores the Rules table as a separate database table on disk.
A modification of this concept, the *BST-RAM* implementation, stores the Rules table completely within main memory. This makes the Rules table management faster but also susceptible to failures like power failure.
The third implementation, called *Fast-BST*, does not use a separate Rules table anymore. Instead, Fast-BST adds an additional column "Condition" of type string to each table, which stores the conditions under which the corresponding data row becomes valid. Thus, conditions need not be derived from the Rules table; each tuple contains its conditions within the "Condition" column. Thus, Fast-BST makes Rules table lookups to derive the conditions under which a tuple becomes valid superfluous, and Fast-BST speeds up write operations that operate on many tuples for the following reason: The database does not need to generate and associate unique IDs to replaced conditions, it can update the "Condition" column in one pass by concatenating its value with the transaction ID. Furthermore, Fast-BST is not susceptible to power failures as BST-RAM as it does not hold the Rules table in its main memory.
In the following, we describe the Fast-BST implementation that is used in the evaluation.

7.1. Fast-BST – Write Operations

Fast-BST implements the concept of BST as follows. Fast-BST stores the before image and the after image of tuples that have been modified by BST transactions. For this purpose, the Fast-BST adds the column "Conditions" to each table R that it uses to construct the relation R", cf. Table 3.
The step of Algorithm 1 is executed for each insert statement `INSERT INTO R <data>` of a transaction T_i. Two equal insert statements are executed twice and are not mixed.

Algorithm 1 Implementing Inserts

1. Insert the tuple (<data>, T_i) into the corresponding table R", and add T_i to the column "Conditions" of the newly inserted tuples.

For each delete statement DELETE ... WHERE <X> of a transaction T_i, the rewriting shown by Algorithm 2 is required.

Algorithm 2 Implementing Deletes

1. Add the substring $\overline{T_i}$ to each entry in the column "Conditions" of each row where <X> evaluates to true. Simplify the Conditions, if T_i wants to delete a tuple that T_i has just inserted before.

Algorithm 3 describes the update operation for the update statement UPDATE ... WHERE <X>.

Algorithm 3 Implementing Updates

1. Copy the tuples for which <X> evaluates to *true* into new data tuples, and concatenate T_i to the existing entries of the "Conditions" attribute of the new tuples. Update the newly copied tuples according to the update statement.
2. Add $\overline{T_i}$ to each entry in the column "Conditions" of each row where <X> evaluates to true and that was not copied and updated in Step 1.

Example 13 *Assume that we execute the following sequence of three transactions T_1, T_2, and T_3, each containing one update statement, on Table 3.*

T_1: *UPDATE Table1 SET Attributes=α_3 WHERE ID=1*
T_2: *UPDATE Table1 SET Attributes=α_4 WHERE ID=2*
T_3: *UPDATE Table1 SET Attributes=α_2*
WHERE (Attributes=$\alpha_3 \vee$ Attributes=α_4)

Table 4 shows the result when all of the distributed transactions $T_1 \ldots T_3$ block and Bi-State-Terminate. Fast-BST marks all tuples changed by transaction T_1 as before image (line (1)), copies them to line (3), and executes the update (on line (3)). Note that Table 4 shows the result when all transactions T_1, T_2, and T_3 block, therefore it already contains the entry for T_3 in line (3). The same algorithm is applied for T_2. When T_3 is executed and both transactions T_1 and T_2 block, T_3 depends on T_1 and T_2. However, FAST-BST does not explicitly check for this dependency. In our example, T_3 only modifies data when either T_1 or T_2 commit. This dependency is maintained automatically by Step 1 of Algorithm 3 since the <condition> of the update statement of T_3 is only true *in lines (3) and (4). Then, these two rows are copied to the rows in line (5) and line (6), and T_3 is added to the "Conditions" column of each of these rows.*

7.2. Fast-BST – Read-Operations

Read operations are modified in the following way: Each value of the returned result additionally contains the corresponding value of the "Conditions" column. Thus, each read operation must be processed by the database only once, regardless of the number of depending blocked transactions. However, the result R is not directly returned to the application, Fast-BST first checks whether the result R contains any entries in the "Conditions" column. If this is the case, it is the application's choice whether it handles these multiple uncertain results, or whether the application delays the read operation until the transactions listed in the "Conditions" column of R have been committed or aborted. If the application can handle multiple results, we can reduce the amount of transferred data by returning an object that represents the different possible valid database states directly within the application by means of the "Conditions" column.

	ID	Attributes	Conditions
(1)	1	α_1	$\overline{T_1}$
(2)	2	α_2	$\overline{T_2}$
(3)	1	α_3	$T_1, \overline{T_3}$
(4)	2	α_4	$T_2, \overline{T_3}$
(5)	1	α_2	T_1, T_3
(6)	2	α_2	T_2, T_3

Table 4. Content of Table 1 after Bi-State-Terminating T_1, T_2, and T_3

7.3. Commit and Abort

The following rules apply when a blocked transaction T_i commits or aborts:
T_i **commits:** Delete all rows that contain $\overline{T_i}$ in the column "Conditions". Delete the string T_i from all entries within the "Conditions" column of the table.
T_i **aborts:** Is treated as $\overline{T_i}$ commits.

Example 14 *Assume T_3 commits. In this case, lines (3) and (4) are deleted from Table 4. Furthermore, the string T_3 must be deleted in Lines (5) and (6) from the attribute values for the column "Conditions", since a commit of T_1 or T_2 automatically implies that the changes of T_3 become valid.*
Note that the data set increases only temporarily and collapses to the original size when the commit decision for the Bi-State-Terminated transactions is known. For example, when the database receives the commit decisions for T_1 and T_3, the database knows the exact unique value for the data tuple with ID 1, which corresponds to Line (5) in case T_3 and T_1 commit, to Line (3) in case T_1 commits and T_3 aborts, and to Line (1) in case T_1 aborts.

8. EXPERIMENTAL EVALUATION

We choose the TPC-C benchmark [14] for generating the test data and transactions. The following questions motivate our experimental evaluation: Which BST implementation is faster? How many transactions can be Bi-State-Terminated until the execution time or the database size for following transactions is unacceptably high? How does BST affect the overall transaction throughput and transaction execution time, when a certain percentage of transactions block?

8.1. BST Stress Test

Having a data hotspot, the used BST implementation may have great influence on the transactions execution times since in this case a lot of transactions depend on each other. As this results in a growth of database states, we have compared the three BST implementations and determined how many blocked transactions per tuple each BST implementation can handle in a stress test. For this stress test, we have generated a database table consisting of a single data tuple. We have sequentially executed a number of database transactions on this table that do not finish, i.e. the transactions do not get a commit decision and thus block. Each of these blocked transactions has incremented or decremented the same data that is initially present. In order to be able to process further transactions, we have used our three BST implementations to terminate each blocked transaction. For this reason, each blocked transaction has doubled the number of possible database states, and thus the number of possible values for the initial tuple grows exponentially for the number of blocked transactions. We have measured the time for processing the $(n+1)^{\text{th}}$ transaction when n transactions are blocked and terminated by BST for each BST implementation.
The y-axis of Figure 4 indicates the required time to process a single update transaction when the number of transactions indicated on the x-axis is terminated by BST. As all of these blocked transactions depend on each other, and all of them are coded to modify the initial tuple, the resulting growth in time and space is exponential. However, as the test indicates, the processing of a transaction when 10 blocked transactions have been terminated by BST does not take a large overhead for the Fast-BST implementation.
Both implementations that use a separate "Rules" table, i.e. BST-Disk and BST-RAM, are significantly slower than the Fast-BST implementation. The reason is that when transactions update a lot of data tuples,

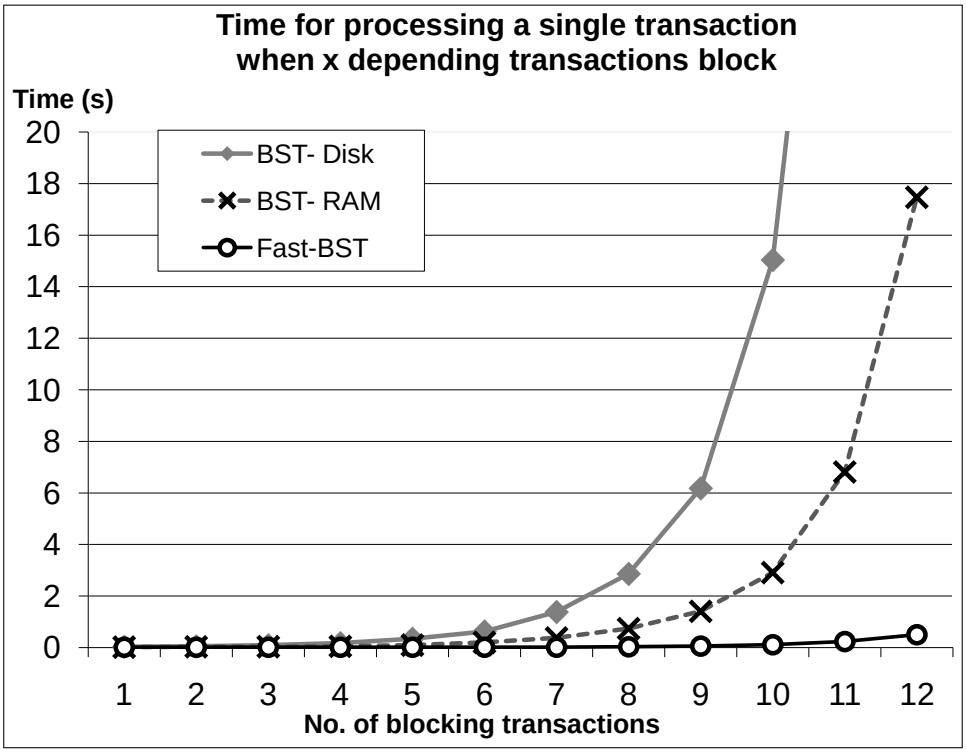

Figure 4. BST Stress Test – Performance

the corresponding condition must be derived from the Rules table for each updated data tuple, and a new condition ID must be generated and assigned separately to each data tuple. In comparison, Fast-BST only adds the transaction's ID to the "Conditions" column of all updated tuples, which can be done much faster. On the y-axis, Figure 5 shows the number of tuples that are generated by BST for a single update operation, while the number of transactions indicated on the x-axis has been terminated by BST. Note that our BST implementations do not differ in the number of resulting tuples. Although the database's size grows exponentially each time a blocked transaction is terminated by BST, it is the database's decision to use BST for a transaction T_i or to wait until the decision for transactions on which T_i depends is known.
As we have seen, using the Fast-BST implementation allows the database to terminate more blocked transactions without a significant loss of performance. For this reason, we use the Fast-BST implementation in the following TPC-C benchmark test.

8.2. BST TPC-C Test

The TPC-C benchmark [14], an online transaction processing benchmark test, simulates an online-shop-like environment in which users execute order transactions against a database. The transactions additionally include recording payments, checking the status of orders, and monitoring the level of stock at the warehouses.
We used a TPC-C "scaling factor" of 2, which results in 139 MB of data and a total amount of 294 transactions, 41,8% of them containing update operations. Characteristic for our implementation of the TPC-C benchmark is that the involved update transactions operate on a set of data tuples whose cardinality is low (i.e. 2 tuples), so we can expect a lot of depending write transactions. In order to simulate transaction blocking, a separate coordinator instance coordinates each transaction and delays the commit command based on different parameters in order to simulate blocked distributed transactions. For example, to simulate a transaction blocking of 1% of all transactions, we delayed the commit command of each 100th transaction. Figure 6 shows the sum of all successfully committed transactions on the y-axis. On the x-axis, the overall time is shown. The different curves indicate whether BST was enabled, and they vary in the percentage of blocked transactions. Note that due to our simulated hotspot, a huge amount of transactions depend on each other. We can see that BST-enabled transaction processing is able to commit a lot more transactions than BST disabled transaction processing.

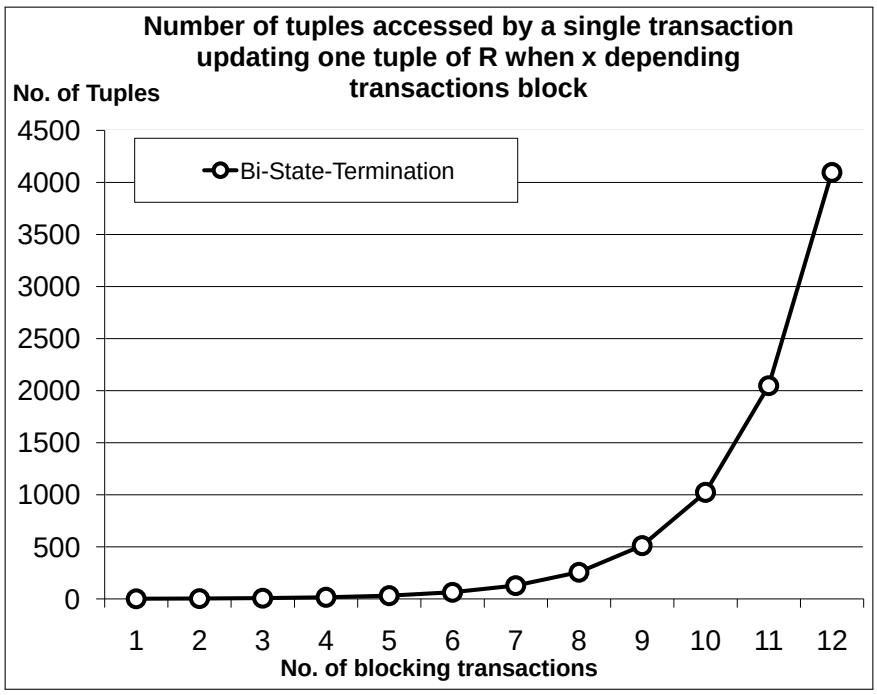

Figure 5. BST Stress Test – Space

Note that the additional space used by BST is rather low, i.e., in our TPC-C experiments, BST requires only about 2% more space.

8.3. Evaluation Summary

We have run a stress test to compare three implementations for BST. While the BST-Disk and BST-RAM implementation use a separate Rules table in order to manage the dependencies of the before- and after-images of the transactions, the Fast-BST implementation directly annotates the rule under which each data tuple becomes valid to the data row. The Fast-BST implementation is able to cope with 10 blocked transactions that all write on the same tuple without causing a loss of performance, while the BST-Disk and BST-RAM implementations can only handle 5 to 7 blocked transactions accessing the same tuple within reasonable time.

When setting up a transaction scenario, two extreme scenarios are possible that influence the outcome of BST: In the first scenario, each transaction operates on different tuples that are not accessed by any other transaction. In the second scenario, each transaction operates on the same tuples. As transaction blocking in the first scenario does not have any influence on other transactions, enabling BST does not commit more transactions than disabling BST. In the second scenario, a blocked transaction would immediately prevent all following transactions from being processed. In this scenario, BST would allow the commitment of almost all transactions, while disabling BST would result in a total blocking situation.

Due to these two possible extreme scenarios, we used the TPC-C benchmark that simulates a typical warehouse environment to get an impression of how BST enhances transaction processing in a real-world scenario. We have preferred the Fast-BST implementation, which is able to enhance the amount of committed transactions in our TPC-C benchmark by 40 to 70%, depending on the number of blocked transactions.

Finally, note that if no more space is available or the required processing time grows, the database can decide for each individual transaction whether to use BST or to wait for the commit decision as in 2PC. In other words, our solution does not force the database to accept long execution times, and BST-enabled transaction processing never blocks more transactions than traditional transaction processing. Our approach can be regarded as a generalization of traditional 2PC in the sense that for each individual transaction executed on a local database, BST is possible but not required.

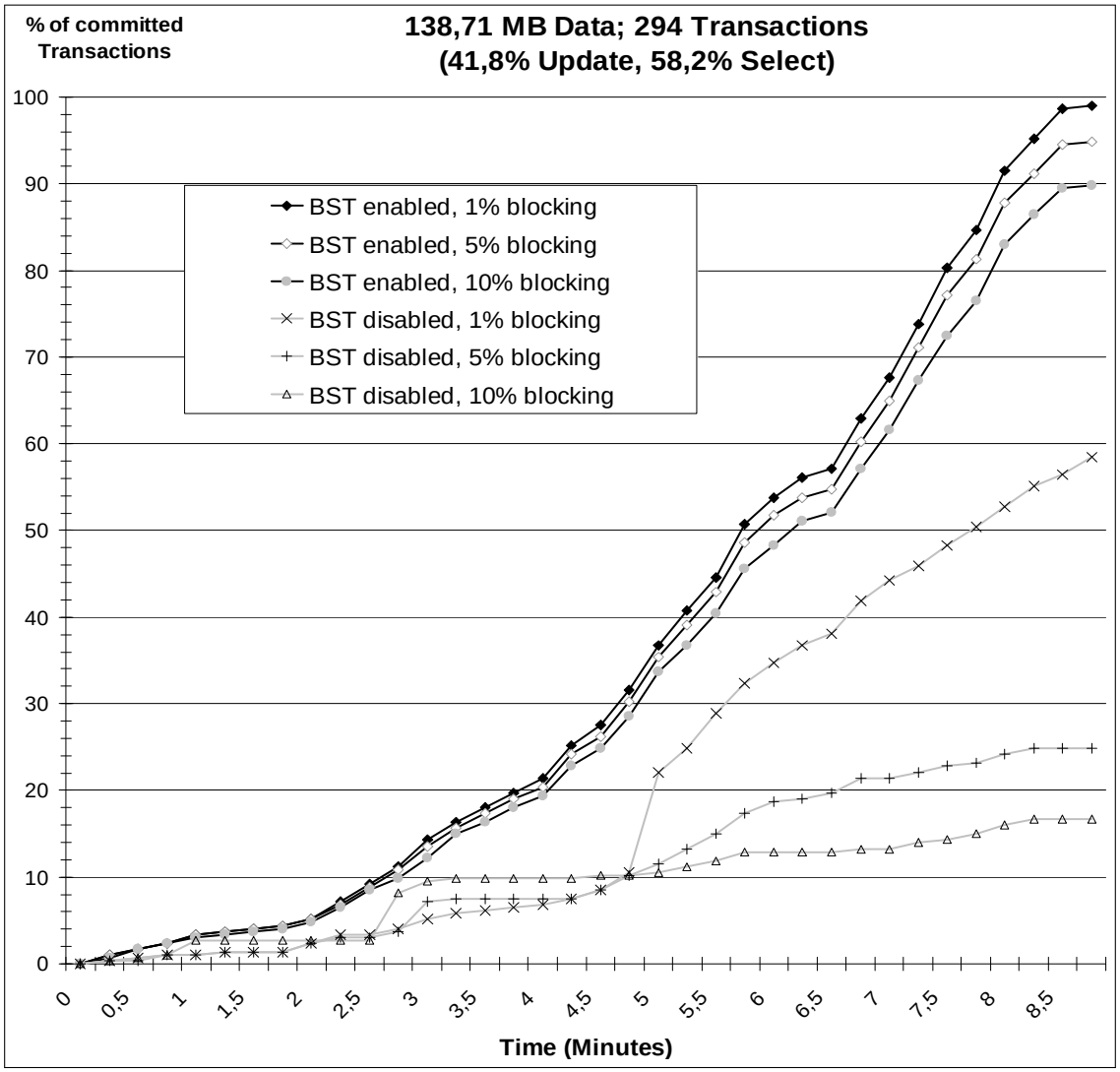

Figure 6. BST evaluation on the TPC-C benchmark

9. RELATED WORK

Concurrency control like multiversion concurrency control [4, 24], timestamp-based concurrency control [17], or optimistic concurrency control [12, 16] omit the use of locks. However, these approaches do not solve the infinite transaction blocking problem on concurrent transactions since each database that has sent the voteCommit message proposes that it will commit the transaction regardless of the used concurrency control mechanism. Therefore, without Bi-State-Termination, the database cannot process a transaction U that is depending on a transaction T, while T waits for the final commit decision, even if the database uses locking-free concurrency control. This motivates the use of BST, which is a termination mechanism that supports the actually used concurrency control mechanism.

Other approaches rely on compensation of transactions. [15], for instance, proposes a timeout-based protocol especially for mobile networks, which requires a compensation of transactions. However, inconsistencies may occur when some databases do not immediately receive the compensation decision or when the coordination process fails.

In order to enhance the availability of the coordination process, some proposals rely on multiple coordinators. [10], for instance, proposes a consensus-based commit protocol that involves multiple coordinators.

However, the problem of *transaction blocking* in the sense of Definition 2, which occurs when the executing database disconnects from the network after sending the voteCommit message, has, to the authors' knowledge, not been studied yet. Even 1PC [1, 2], which does not require a vote message but acknowledges each

operation, encounters the problem of transaction blocking since each acknowledged operation that accesses a data tuple must block this data tuple until the transaction is successfully completed.
Our solution relates to three ideas that are used in different contexts: Escrow locks [11], speculative locking [21], and multiversion databases [6, 7, 13].
Escrow locks are a refinement of field calls, which are used in environments where data hotspots are frequently accessed. The escrow lock calculates an interval $[i, k]$ for an attribute a by means of the currently processed updates. The interval indicates the actual upper and lower boundary that the attribute a may take. When a further transaction relies on a precondition for a, the database checks whether the precondition evaluates to *true* for each value of a that is contained in the interval $[i, k]$. In contrast to the escrow locking technique, Bi-State-Termination, is a transaction termination mechanism. BST neither relies on numerical values, nor assumes that an attribute value must lie in a given interval. BST always knows the exact values that an attribute can actually have and even allows an application to decide that a transaction T_i may only be committed in a certain constellation of commit and abort decisions of transactions on which T_i depends.
Another related locking mechanism is Speculative Locking (SL) [21]. SL was proposed to speed up transaction processing by spawning multiple parallel executions of a transaction that waits for the acquisition of required locks. SL has in common with Bi-Sate-Termination that SL also allows a transaction T_c to access the after-image of a transaction T_i while T_i is waiting for its commit decision. However, unlike Bi-State-Termination, SL does not allow committing T_c before the final commit decision for T_i has been received. This means, SL cannot successfully terminate T_c while the commit vote for T_i is missing. For this reason, SL cannot be used to solve the infinite transaction blocking problem that may occur in mobile networks. Furthermore, our Fast-BST implementation can execute read-operations in one pass even if they return multiple result values due to transactions that wait for the commit decision.
Multiversion database systems [6, 7, 13] are used to support different expressions of a data object. They are used for CAD modelling and versioning systems. However, compared to BST, multiversion database systems allow multiple versions to be concurrently valid, while BST allows only one valid version, but lacks the knowledge which of the multiple versions is valid due to the atomic commit protocol. Whenever BST requires multiple transaction executions that all return the same result, BST is even transparent to the application. Furthermore, multiversion database systems are mostly central embedded databases that are not designed to deal with distributed transactions. Instead, the user explicitly specifies on which version he wants to work.
Complementing this contribution, [19] describes how arbitrary database constraints beyond functional dependencies and referential integrity constraints, can be checked on the BST implementation. Furthermore, [19] suggests a different treatment for row-based integrity constraints and general consistency constraints.

10. SUMMARY AND CONCLUSION

Transaction blocking occurs very frequently, i.e. a sub-transaction has voted for commit, but has not received the commit decision, yet. We argue that the risk of infinite transaction blocking, which can occur if the database moves or disconnects, is not appropriately solved by other approaches to distributed transaction processing. We have explained the concept of Bi-State-Termination that is useful to terminate blocked transactions without violating atomicity, even without knowing the explicit coordinator decision on commit or abort. We have described three different implementations and have experimentally evaluated them.
Our experimental results have shown that Bi-State-Termination enhances the number of committed transactions and that BST is able to deal with a large number of depending blocked transactions without experiencing significant performance loss. This justifies using BST in mobile ad-hoc networks that are exposed to the risk of transaction blocking.
To summarize, we consider Bi-State-Termination as a useful option that is usable for mobile networks in order to terminate a transaction instead of just waiting for the commit decision for a long time.

REFERENCES

[1] M. Abdallah, R. Guerraoui, and P. Pucheral. One-phase commit: Does it make sense? In *ICPADS '98: Proceedings of the 1998 International Conference on Parallel and Distributed Systems*, page 182, Washington, DC, USA, 1998. IEEE Computer Society.

[2] Y. J. Al-Houmaily and P. K. Chrysanthis. 1-2pc: the one-two phase atomic commit protocol. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC), Nicosia, Cyprus, March 14-17*, pages 684–691, 2004.

[3] R. Bayer, H. Heller, and A. Reiser. Parallelism and recovery in database systems. *ACM Trans. Database Syst.*, 5(2):139–156, 1980.

[4] P. A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] W. Cellary and G. Jomier. Consistency of versions in object-oriented databases. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 432–441. Morgan Kaufmann, 1990.

[7] I.-M. A. Chen, V. M. Markowitz, S. Letovsky, P. Li, and K. H. Fasman. Version management for scientific databases. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, volume 1057 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 1996.

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[9] J. Gray. Notes on data base operating systems. In M. J. Flynn, J. Gray, A. K. Jones, et al., editors, *Advanced Course: Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer, 1978.

[10] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[12] T. Haerder. Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9(2):111–120, 1984.

[13] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv.*, 22(4):375–409, 1990.

[14] W. Kohler, A. Shah, and F. Raab. Overview of TPC Benchmark C: The Order-Entry Benchmark. Technical report, `http://www.tpc.org`, Transaction Processing Performance Council, 1991.

[15] V. Kumar, N. Prabhu, M. H. Dunham, and A. Y. Seydim. Tcot - a timeout-based mobile transaction commitment protocol. *IEEE Transactions on Computers*, 51(10):1212–1218, 2002.

[16] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[17] P.-J. Leu and B. K. Bhargava. Multidimensional timestamp protocols for concurrency control. In *Proceedings of the Second International Conference on Data Engineering*, pages 482–489, Washington, DC, USA, 1986. IEEE Computer Society.

[18] S. Obermeier and S. Böttcher. Avoiding infinite blocking of mobile transactions. In *Proceedings of the 11th International Database Engineering & Applications Symposium (IDEAS), Banff, Canada*, 2007.

[19] S. Obermeier and S. Böttcher. Constraint checking for non-blocking transaction processing in mobile ad-hoc networks. In *Proceedings of the 12th International Conference on Enterprise Information Systems (ICEIS), Funchal, Madeira - Portugal*, 2010.

[20] P. K. Reddy and M. Kitsuregawa. Reducing the blocking in two-phase commit with backup sites. *Inf. Process. Lett.*, 86(1):39–47, 2003.

[21] P. K. Reddy and M. Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 16(2):154–169, 2004.

[22] D. Skeen. Nonblocking commit protocols. In Y. E. Lien, editor, *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan*, pages 133–142. ACM Press, 1981.

[23] M. R. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. In *Distributed systems, Vol. II: distributed data base systems*, pages 193–199, Norwood, MA, USA, 1986. Artech House, Inc.

[24] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

Authors

Sebastian Obermeier works as a Scientist at ABB Corporate Research. He holds a PhD and Diploma in Computer Science from the University of Paderborn, Germany. His research areas include the use of database transaction technology within mobile ad-hoc networks and security for critical infrastructures.

Stefan Böttcher is a Professor of Computer Science at the University of Paderborn. His research areas cover query optimization and compression in XML databases, transactions in mobile ad-hoc networks, security and privacy. Before he joined the University of Paderborn, he worked for several years at the German research labs of IBM and Daimler Benz.